[7]:
```python
# Import libraries
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
import itertools
import argparse
import sys
import time
from sklearn import preprocessing
import pandas as pd
import os
os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
```

[8]:
```python
## Preprocessing of data
# Function to load data

def get_power_data():
    """
    Read the Individual household electric power consumption dataset
    """

    # Assume that the dataset is located on folder "data"
    data = pd.read_csv('household_power_consumption.txt',
                       sep=';', low_memory=False)

    # Drop some non-predictive variables
    data = data.drop(columns=['Date', 'Time'], axis=1)

    #print(data.head())

    # Replace missing values
    data = data.replace('?', np.nan)

    # Drop NA
    data = data.dropna(axis=0)

    # Normalize
    standard_scaler = preprocessing.StandardScaler()
    np_scaled = standard_scaler.fit_transform(data)
    data = pd.DataFrame(np_scaled)

    # Goal variable assumed to be the first
    X = data.values[:, 1:].astype('float32')
    y = data.values[:, 0].astype('float32')

    # Create categorical y for binary classification with balanced classes
    y = np.sign(y+0.46)
```

```
    # Split train and test data here: (X_train, Y_train, X_test, Y_test)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
    no_class = 2                    #binary classification

    return X_train, X_test, y_train, y_test, no_class
```

[9]:
```
X_train, X_test, y_train, y_test, no_class = get_power_data()
print("X,y types: {} {}".format(type(X_train), type(y_train)))
print("X size {}".format(X_train.shape))
print("Y size {}".format(y_train.shape))

# Create a binary variable from one of the columns.
# You can use this OR not

idx = y_train >= 0
notidx = y_train < 0
y_train[idx] = 1
y_train[notidx] = -1
```

```
X,y types: <class 'numpy.ndarray'> <class 'numpy.ndarray'>
X size (1536960, 6)
Y size (1536960,)
```

[10]:
```
# Sigmoid function
def sigmoid(x, derivative=False):
    sigm = 1. / (1. + np.exp(-x))
    if derivative:
        return sigm * (1. - sigm)
    return sigm

# Define weights initialization
def initialize_w(N, d):
    return 2*np.random.random((N,d)) - 1
```

[11]:
```
# Fill in feed forward propagation
def feed_forward_propagation(X, y, w_1, w_2, w_3, lmbda):
    # Fill in
    layer_0 = X.T
    layer_1 = sigmoid(np.dot(w_1.T,layer_0))
    layer_2 = sigmoid(np.dot(w_2.T,layer_1))
    layer_3 = np.dot(w_3.T,layer_2)
    return layer_0, layer_1, layer_2, layer_3
```

[12]:
```
## Fill in backpropagation
def back_propagation(y, w_1, w_2, w_3, layer_0, layer_1, layer_2, layer_3):
    # Calculate the gradient here
    # Compute the error
```

```
    d_layer_3 = layer_3 - y.reshape(1,y.shape[0])
    d_layer_2 = np.dot(w_3, d_layer_3) * layer_2 * (1 - layer_2)
    d_layer_1 = np.dot(w_2, d_layer_2) * layer_1 * (1 - layer_1)

    # Compute the gradients of the weights
    layer_3_delta = np.dot(layer_2, d_layer_3.T)
    layer_2_delta = np.dot(layer_1, d_layer_2.T)
    layer_1_delta = np.dot(layer_0, d_layer_1.T)

    return layer_1_delta, layer_2_delta, layer_3_delta
```

[13]:
```
# Cost function
def cost(X, y, w_1, w_2, w_3, lmbda):
    N, d = X.shape
    a1,a2,a3,a4 = feed_forward_propagation(X,y,w_1,w_2,w_3,lmbda)
    #regularization = (lmbda) * ((np.linalg.norm(w_1)** 2) +( np.linalg.
 ↪norm(w_2) ** 2) + (np.linalg.norm(w_3)** 2))
    #return regularization + np.linalg.norm(a4[:,0] - y,2) ** 2 / N #with␣
 ↪regularization
    return np.linalg.norm(a4[:,0] - y,2) ** 2 / N  #without regularization
```

[14]:
```
def SGD(X, y, w_1, w_2, w_3, lmbda, learning_rate, batch_size):
    N = X.shape[0]
    data_index = np.random.choice(N, batch_size)  #Random choose the data i,␣
 ↪using batch from [i,i+batch_size-1]


    X_batch = X[data_index, :]
    y_batch = y[data_index]

    # Forward pass
    layer_0, layer_1, layer_2, layer_3 = feed_forward_propagation( X_batch,␣
 ↪y_batch, w_1, w_2, w_3, lmbda)

    # Backward pass
    d_w_1, d_w_2, d_w_3 = back_propagation(y_batch, w_1, w_2, w_3, layer_0,␣
 ↪layer_1, layer_2, layer_3)


    w_3 -= learning_rate * (d_w_3 + lmbda * w_3 / N)
    w_2 -= learning_rate * (d_w_2 + lmbda * w_2 / N)
    w_1 -= learning_rate * (d_w_1 + lmbda * w_1 / N)

    return w_1, w_2, w_3
```

[44]:
```python
def SVRG(X, y, w_1, w_2, w_3, lmbda, learning_rate, batch_size, T):

    N = X.shape[0]
    m = int(N / batch_size)
    # Initialize weights
    w_3_last = w_3.copy()
    w_2_last = w_2.copy()
    w_1_last = w_1.copy()

    for t in range(T):
        # Full gradient
        layer_0, layer_1, layer_2, layer_3 = feed_forward_propagation(X, y,␣
↪w_1_last, w_2_last, w_3_last, lmbda)
        layer_1_delta, layer_2_delta, layer_3_delta = back_propagation(y,␣
↪w_1_last, w_2_last, w_3_last, layer_0, layer_1, layer_2, layer_3)

        # Randomly choose a data point from the batch
        for i in range(m):
            # Choose random data point from batch
            batch_indices = np.random.choice(N, batch_size)
            X_batch = X[batch_indices, :]
            y_batch = y[batch_indices]

            # Compute stochastic gradient
            c1, c2, c3, c4 = feed_forward_propagation(X_batch, y_batch,␣
↪w_1_last, w_2_last, w_3_last, lmbda)
            grad_zeta_w_1_tilde, grad_zeta_w_2_tilde, grad_zeta_w_3_tilde =␣
↪back_propagation(y_batch, w_1_last, w_2_last, w_3_last, c1, c2, c3, c4)

            # Compute full gradient at the same data point
            a1_zeta, a2_zeta, a3_zeta, a4_zeta =␣
↪feed_forward_propagation(X_batch, y_batch, w_1_last, w_2_last, w_3_last, lmbda)
            layer_1_delta_zeta, layer_2_delta_zeta, layer_3_delta_zeta =␣
↪back_propagation(y_batch, w_1_last, w_2_last, w_3_last, a1_zeta, a2_zeta,␣
↪a3_zeta, a4_zeta)

            # Compute variance reduced gradient
            b1, b2, b3, b4 = feed_forward_propagation(X_batch, y_batch,␣
↪w_1_last, w_2_last, w_3_last, lmbda)
            grad_zeta_w_1, grad_zeta_w_2, grad_zeta_w_3 =␣
↪back_propagation(y_batch, w_1_last, w_2_last, w_3_last,b1, b2, b3, b4)
            grad_w_3 = grad_zeta_w_3 - grad_zeta_w_3_tilde + layer_3_delta_zeta
            grad_w_2 = grad_zeta_w_2 - grad_zeta_w_2_tilde + layer_2_delta_zeta
            grad_w_1 = grad_zeta_w_1 - grad_zeta_w_1_tilde + layer_1_delta_zeta

            # Update weights
```

```
            w_3 -= learning_rate *  (grad_w_3 + lmbda * w_3)/N
            w_2 -= learning_rate * (grad_w_2 + lmbda * w_2)/N
            w_1 -= learning_rate * (grad_w_1 + lmbda * w_1)/N

        # Update last weights after each epoch
        w_3_last = w_3.copy()
        w_2_last = w_2.copy()
        w_1_last = w_1.copy()

    return w_1, w_2, w_3
```

[16]:
```
# Define GD here:
def GD(X, y, w_1,w_2,w_3, learning_rate, lmbda):
    # Complete here:
    N = X.shape[0]
    # Forward pass
    layer_0, layer_1, layer_2, layer_3 = feed_forward_propagation(X, y, w_1,␣
 ↪w_2, w_3, lmbda)
    # Backward pass
    d_w_1, d_w_2, d_w_3 = back_propagation(y, w_1, w_2, w_3, layer_0, layer_1,␣
 ↪layer_2, layer_3)

    w_3 -= learning_rate * (d_w_3 + lmbda * w_3)/N
    w_2 -= learning_rate * (d_w_2 + lmbda * w_2)/N
    w_1 -= learning_rate * (d_w_1 + lmbda * w_1)/N
    return w_1, w_2, w_3
```

[17]:
```
# Define projected GD here:
def PGD(X, y, w_1,w_2,w_3, learning_rate, lmbda, noise):
    # Complete here:
     # Initialize weights
    N = X.shape[0]
    # Forward pass
    layer_0, layer_1, layer_2, layer_3 = feed_forward_propagation(X, y, w_1,␣
 ↪w_2, w_3, lmbda)
    # Backward pass
    d_w_1, d_w_2, d_w_3 = back_propagation(y, w_1, w_2, w_3, layer_0, layer_1,␣
 ↪layer_2, layer_3)

    w_3 -= learning_rate * (d_w_3 + lmbda * w_3)/N + noise*np.random.randn(*w_3.
 ↪shape)
    w_2 -= learning_rate * (d_w_2 + lmbda * w_2)/N + noise*np.random.randn(*w_2.
 ↪shape)
    w_1 -= learning_rate * (d_w_1 + lmbda * w_1)/N + noise*np.random.randn(*w_1.
 ↪shape)
    return w_1, w_2, w_3
```

[18]:
```python
# Define BCD here:
def BCD(X, y, w_1,w_2,w_3, learning_rate, lmbda):
    # Randomly choose one of w_1, w_2, w_3 to update
    N = X.shape[0]
    layer_0, layer_1, layer_2, layer_3 = feed_forward_propagation(X, y, w_1,␣
 ↪w_2, w_3, lmbda)
    # Backward pass
    d_w_1, d_w_2, d_w_3 = back_propagation(y, w_1, w_2, w_3, layer_0, layer_1,␣
 ↪layer_2, layer_3)
    if np.random.choice(3)== 0:
        w_3 -= learning_rate  * (d_w_3 + lmbda * w_3)/N
    if np.random.choice(3)== 1:
        w_2 -= learning_rate  * (d_w_2 + lmbda * w_2)/N
    if np.random.choice(3)== 2:
        w_1 -= learning_rate  * (d_w_1 + lmbda * w_1)/N

    return w_1, w_2, w_3
```

[45]:
```python
y_train
```

[45]:
```
array([ 1., -1.,  1., ...,  1.,  1., -1.], dtype=float32)
```

[46]:
```python
# Should be a hyperparameter that you tune, not an argument - Fill in the values
parser = argparse.ArgumentParser()

# Power
parser.add_argument('--lambda', type=float, default=1e-2, dest='lmbda')
parser.add_argument('--w_size', type=int, default=50, dest='w_size')
parser.add_argument('--lr', type=float, default=1e-2)
parser.add_argument('--iterations', type=int, default=100)
args = parser.parse_args([])

#args, unknown_args = parser.parse_known_args()
```

[47]:
```python
def defineOptimizer(i, X_train, y_train, w_1, w_2, w_3, lmbda, lr,␣
 ↪batch_size=16):
    if i == 0:
        optimizers = {# Fill in the hyperparameters
                        "opt": SGD(X_train, y_train, w_1, w_2, w_3, args.lmbda,␣
 ↪args.lr, batch_size=16),
                        "name": "SGD",
                        #"inner": None
                     }
    elif i == 1:
        optimizers = {# Fill in the hyperparameters
                        "opt": SVRG(X_train, y_train, w_1, w_2, w_3, args.lmbda,␣
 ↪args.lr,batch_size=16,T=4),
```

```
                            "name": "SVRG",
                            #"inner": None
                        }
    elif i == 2:
        optimizers = {# Fill in the hyperparameters
                            "opt": GD(
                                X_train, y_train, w_1, w_2, w_3, learning_rate=args.
↪lr,
                                lmbda=args.lmbda),
                            "name": "GD",
                            #"inner": None
                         }
    elif i == 3:
        optimizers = {# Fill in the hyperparameters
                            "opt": PGD(
                                X_train, y_train, w_1, w_2, w_3, learning_rate=args.
↪lr,
                                lmbda=args.lmbda, noise=1e-3),
                            "name": "PGD",
                            #"inner":None
                        }
    elif i == 4:
        optimizers = {# Fill in the hyperparameters
                            "opt": BCD(
                                X_train, y_train, w_1, w_2, w_3, learning_rate=args.
↪lr,
                                lmbda=args.lmbda),
                            "name": "BCD",
                            #"inner": None
                        }
    return optimizers
```

[48]:

```
# Initialize weights
#w_1 = initialize_w(X_train.shape[1], args.w_size)
#w_2 = initialize_w(args.w_size,args.w_size)
#w_3 = initialize_w(args.w_size, 1)

# Get iterations
iterations = args.iterations
```

[42]:

```
# Run the iterates over the algorithms above
loss_= np.zeros((iterations,5))
ti= np.zeros((iterations,5))
algoNames = ['SGD', 'SVRG', 'GD', 'PGD', 'BCD']
```

[59]:

```
i=0
print("Running {}".format(algoNames[i]))
```

```
# initialize weights
w_1 = initialize_w(X_train.shape[1], args.w_size)
w_2 = initialize_w(args.w_size,args.w_size)
w_3 = initialize_w(args.w_size, 1)

for j in range(iterations):
    print("......",j,".......")
    start = time.time()
    optimizers = defineOptimizer(i, X_train, y_train, w_1, w_2, w_3, args.lmbda, args.lr)
    end = time.time()
    loss_[j,i] = cost(X_train, y_train, w_1, w_2, w_3, args.lmbda)
    ti[j,i] = end-start
    print("......",loss_[j,i],".......")
```

```
Running SGD
... 0 ...
... 56.46027697475826 ...
... 1 ...
... 163.91462797092308 ...
... 2 ...
... 122.49368781465586 ...
... 3 ...
... 2.7498753382993684 ...
... 4 ...
... 2.1148537457614736 ...
... 5 ...
... 1.21860518087522 ...
... 6 ...
... 1.0294934942020668 ...
... 7 ...
... 1.0917609586467836 ...
... 8 ...
... 1.0555748415996022 ...
... 9 ...
... 1.103825726886375 ...
... 10 ...
... 1.0845586368200637 ...
... 11 ...
... 1.1056206677932812 ...
... 12 ...
... 1.0738424869337495 ...
... 13 ...
... 1.0578268316805313 ...
... 14 ...
... 1.0432472952243774 ...
... 15 ...
```

```
... 1.050117073449813 ...
... 16 ...
... 1.0197018749385725 ...
... 17 ...
... 1.006704879883306 ...
... 18 ...
... 1.0123768445367682 ...
... 19 ...
... 1.006622481477758 ...
... 20 ...
... 1.018734105989211 ...
... 21 ...
... 1.0389971188287759 ...
... 22 ...
... 1.0441605800946265 ...
... 23 ...
... 1.1023047737716551 ...
... 24 ...
... 1.1157755233220819 ...
... 25 ...
... 1.1068914559840555 ...
... 26 ...
... 1.0787557286557818 ...
... 27 ...
... 1.0293044053701723 ...
... 28 ...
... 1.0068744850016966 ...
... 29 ...
... 1.0123911128793166 ...
... 30 ...
... 1.0288074538005305 ...
... 31 ...
... 1.0573810669441632 ...
... 32 ...
... 1.1472009115610264 ...
... 33 ...
... 1.1243786631270938 ...
... 34 ...
... 1.1341106600252 ...
... 35 ...
... 1.0733952694550004 ...
... 36 ...
... 1.1666188818456893 ...
... 37 ...
... 1.1470528718162758 ...
... 38 ...
... 1.1697669579154863 ...
... 39 ...
```

```
... 1.2246759412776278 ...
... 40 ...
... 1.225918632333549 ...
... 41 ...
... 1.1409370977284994 ...
... 42 ...
... 1.1595119668908476 ...
... 43 ...
... 1.260578314491524 ...
... 44 ...
... 1.2493382006745892 ...
... 45 ...
... 1.2801024155293173 ...
... 46 ...
... 1.2686125591288984 ...
... 47 ...
... 1.2166666137264968 ...
... 48 ...
... 1.1936441609206332 ...
... 49 ...
... 1.2744791965970075 ...
... 50 ...
... 1.256877070782489 ...
... 51 ...
... 1.1772433259514354 ...
... 52 ...
... 1.1742520894445077 ...
... 53 ...
... 1.2334185376623772 ...
... 54 ...
... 1.2943377447089617 ...
... 55 ...
... 1.2981257479056831 ...
... 56 ...
... 1.272122094462739 ...
... 57 ...
... 1.2166889163880539 ...
... 58 ...
... 1.150536709603279 ...
... 59 ...
... 1.2392214876942302 ...
... 60 ...
... 1.3404601844810866 ...
... 61 ...
... 1.2356819749044785 ...
... 62 ...
... 1.290898848631962 ...
... 63 ...
```

... 1.2628553141212233 ...
... 64 ...
... 1.3740165044313664 ...
... 65 ...
... 1.4221450583659574 ...
... 66 ...
... 1.378722713689454 ...
... 67 ...
... 1.203776686276562 ...
... 68 ...
... 1.341433379794002 ...
... 69 ...
... 1.2782872619741092 ...
... 70 ...
... 1.353658662502284 ...
... 71 ...
... 1.281578995396923 ...
... 72 ...
... 1.2240506411964338 ...
... 73 ...
... 1.3010436826031901 ...
... 74 ...
... 1.3089268211152871 ...
... 75 ...
... 1.3161515990848633 ...
... 76 ...
... 1.3636200856457619 ...
... 77 ...
... 1.3680535609239548 ...
... 78 ...
... 1.275264013776728 ...
... 79 ...
... 1.4460096243964047 ...
... 80 ...
... 1.40545027993551 ...
... 81 ...
... 1.3573774080397496 ...
... 82 ...
... 1.271130667940872 ...
... 83 ...
... 1.2671969514131538 ...
... 84 ...
... 1.3334351369038988 ...
... 85 ...
... 1.4185759295779758 ...
... 86 ...
... 1.378327612751623 ...
... 87 ...

```
... 1.3574632312283685 ...
... 88 ...
... 1.3389500665432874 ...
... 89 ...
... 1.3259855189882523 ...
... 90 ...
... 1.4108537898769762 ...
... 91 ...
... 1.3525689153554177 ...
... 92 ...
... 1.396028356023615 ...
... 93 ...
... 1.3008669468378153 ...
... 94 ...
... 1.3282770515415674 ...
... 95 ...
... 1.3134719122659593 ...
... 96 ...
... 1.4044078404435272 ...
... 97 ...
... 1.4043376198667745 ...
... 98 ...
... 1.4884885317336622 ...
... 99 ...
... 1.497060141030746 ...
```

[72]:
```
i=1
algoNames = ['SGD', 'SVRG', 'GD', 'PGD', 'BCD']
print("Running {}".format(algoNames[i]))
# initialize weights
w_1 = initialize_w(X_train.shape[1], args.w_size)
w_2 = initialize_w(args.w_size,args.w_size)
w_3 = initialize_w(args.w_size, 1)

for j in range(iterations):
    print("......",j,".......")
    start = time.time()
    optimizers = defineOptimizer(i, X_train, y_train, w_1, w_2, w_3, args.lmbda,
 ↪10*args.lr)
    end = time.time()
    loss_[j,i] = cost(X_train, y_train, w_1, w_2, w_3, args.lmbda)
    ti[j,i] = end-start
    print("......",loss_[j,i],".......")
```

```
Running SVRG
... 0 ...
... 5.801811129491061 ...
... 1 ...
```

```
... 1.7658362959932774 ...
... 2 ...
... 1.1169195695189478 ...
... 3 ...
... 1.0145096623392247 ...
... 4 ...
... 1.0005854817730915 ...
... 5 ...
... 1.0004500243277963 ...
... 6 ...
... 1.0021356254273026 ...
... 7 ...
... 1.0038528853734017 ...
... 8 ...
... 1.0055141754174475 ...
... 9 ...
... 1.0071339404873203 ...
... 10 ...
... 1.008896029596423 ...
... 11 ...
... 1.0107052573301982 ...
... 12 ...
... 1.0125423043455664 ...
... 13 ...
... 1.014365307755677 ...
... 14 ...
... 1.0162139537101114 ...
... 15 ...
... 1.0181502449418147 ...
... 16 ...
... 1.02001294498781 ...
... 17 ...
... 1.0219385136701937 ...
... 18 ...
... 1.0238748054994056 ...
... 19 ...
... 1.0258438966775476 ...
... 20 ...
... 1.0276407443276827 ...
... 21 ...
... 1.029509625643607 ...
... 22 ...
... 1.0314065002879933 ...
... 23 ...
... 1.0331776388185148 ...
... 24 ...
... 1.034942124833767 ...
... 25 ...
```

... 1.0365730678250427 ...
... 26 ...
... 1.0382235396978194 ...
... 27 ...
... 1.0397478105881184 ...
... 28 ...
... 1.0412378247438365 ...
... 29 ...
... 1.0427605528562092 ...
... 30 ...
... 1.0443664484109245 ...
... 31 ...
... 1.0456909388775677 ...
... 32 ...
... 1.0469261297263388 ...
... 33 ...
... 1.0482816251019502 ...
... 34 ...
... 1.0495130821914644 ...
... 35 ...
... 1.050566904107634 ...
... 36 ...
... 1.0516905303687962 ...
... 37 ...
... 1.052751289908657 ...
... 38 ...
... 1.0536688764960391 ...
... 39 ...
... 1.0546947643224738 ...
... 40 ...
... 1.0555604829092202 ...
... 41 ...
... 1.0564370539035088 ...
... 42 ...
... 1.05703261339481 ...
... 43 ...
... 1.0580348062376275 ...
... 44 ...
... 1.058606467363283 ...
... 45 ...
... 1.0592966767415521 ...
... 46 ...
... 1.0599136735403745 ...
... 47 ...
... 1.0604354236445592 ...
... 48 ...
... 1.061056344494528 ...
... 49 ...

... 1.061553716616882 ...
... 50 ...
... 1.062073665272869 ...
... 51 ...
... 1.0624398983798125 ...
... 52 ...
... 1.0627592845411764 ...
... 53 ...
... 1.0632905021507841 ...
... 54 ...
... 1.0635307217847862 ...
... 55 ...
... 1.064003466878258 ...
... 56 ...
... 1.0641705809982875 ...
... 57 ...
... 1.0644710239213593 ...
... 58 ...
... 1.0645138292306502 ...
... 59 ...
... 1.0647334465841336 ...
... 60 ...
... 1.0648247371093729 ...
... 61 ...
... 1.0649807081952767 ...
... 62 ...
... 1.065303728461684 ...
... 63 ...
... 1.065377186152193 ...
... 64 ...
... 1.0653864864626847 ...
... 65 ...
... 1.0653820743402942 ...
... 66 ...
... 1.0655033279130182 ...
... 67 ...
... 1.0655729981441786 ...
... 68 ...
... 1.06565280724658 ...
... 69 ...
... 1.0655707793453593 ...
... 70 ...
... 1.0655490462617503 ...
... 71 ...
... 1.0656621600297824 ...
... 72 ...
... 1.0655063776332878 ...
... 73 ...

... 1.0655591658957881 ...
... 74 ...
... 1.0654518229112317 ...
... 75 ...
... 1.065401190676526 ...
... 76 ...
... 1.0653575994825688 ...
... 77 ...
... 1.0653044342990885 ...
... 78 ...
... 1.0651047644835716 ...
... 79 ...
... 1.065121329955825 ...
... 80 ...
... 1.065032657006629 ...
... 81 ...
... 1.0647867893424618 ...
... 82 ...
... 1.0645630262713597 ...
... 83 ...
... 1.0644118559482612 ...
... 84 ...
... 1.0643781195760835 ...
... 85 ...
... 1.0643543536808802 ...
... 86 ...
... 1.0641377142533563 ...
... 87 ...
... 1.063919679563854 ...
... 88 ...
... 1.063787257685499 ...
... 89 ...
... 1.0635739607364576 ...
... 90 ...
... 1.0634428451644944 ...
... 91 ...
... 1.063134607775308 ...
... 92 ...
... 1.063050543947978 ...
... 93 ...
... 1.062914226619462 ...
... 94 ...
... 1.062861107564531 ...
... 95 ...
... 1.062572269376835 ...
... 96 ...
... 1.0623377022011673 ...
... 97 ...

```
... 1.0620478266097477 ...
... 98 ...
... 1.0619752612563524 ...
... 99 ...
... 1.061799010294133 ...
```

[57]:
```python
i=2
print("Running {}".format(algoNames[i]))
# initialize weights
w_1 = initialize_w(X_train.shape[1], args.w_size)
w_2 = initialize_w(args.w_size,args.w_size)
w_3 = initialize_w(args.w_size, 1)

for j in range(iterations):
    print("......",j,".......")
    start = time.time()
    optimizers = defineOptimizer(i, X_train, y_train, w_1, w_2, w_3, args.lmbda,␣
 ↪args.lr)
    end = time.time()
    loss_[j,i] = cost(X_train, y_train, w_1, w_2, w_3, args.lmbda)
    ti[j,i] = end-start
    print("......",loss_[j,i],".......")
```

```
Running GD
... 0 ...
... 11.688688826434522 ...
... 1 ...
... 7.896676441280257 ...
... 2 ...
... 5.51375704390067 ...
... 3 ...
... 4.005435378565351 ...
... 4 ...
... 3.0426199929592954 ...
... 5 ...
... 2.4219425261308567 ...
... 6 ...
... 2.0172896116846695 ...
... 7 ...
... 1.750148829463243 ...
... 8 ...
... 1.5714068403805448 ...
... 9 ...
... 1.4501550471394933 ...
... 10 ...
... 1.3667945235093495 ...
... 11 ...
```

```
... 1.3087822199104555 ...
... 12 ...
... 1.2680012332499602 ...
... 13 ...
... 1.239130522912368 ...
... 14 ...
... 1.218630399920149 ...
... 15 ...
... 1.2041080551456271 ...
... 16 ...
... 1.1939182176119882 ...
... 17 ...
... 1.1869097845716188 ...
... 18 ...
... 1.182263485341644 ...
... 19 ...
... 1.179386652076209 ...
... 20 ...
... 1.1778440813173507 ...
... 21 ...
... 1.177311913685064 ...
... 22 ...
... 1.1775463562105373 ...
... 23 ...
... 1.1783620993885737 ...
... 24 ...
... 1.1796171597400908 ...
... 25 ...
... 1.1812020503156135 ...
... 26 ...
... 1.1830319168380736 ...
... 27 ...
... 1.1850407421843023 ...
... 28 ...
... 1.187177018936883 ...
... 29 ...
... 1.189400481498406 ...
... 30 ...
... 1.191679614894307 ...
... 31 ...
... 1.1939897408363775 ...
... 32 ...
... 1.1963115379918914 ...
... 33 ...
... 1.1986298922126586 ...
... 34 ...
... 1.2009329996149976 ...
... 35 ...
```

... 1.2032116647360647 ...
... 36 ...
... 1.2054587500359446 ...
... 37 ...
... 1.2076687432968618 ...
... 38 ...
... 1.2098374171791237 ...
... 39 ...
... 1.2119615609951497 ...
... 40 ...
... 1.2140387691920986 ...
... 41 ...
... 1.216067274410168 ...
... 42 ...
... 1.2180458156380072 ...
... 43 ...
... 1.2199735340167877 ...
... 44 ...
... 1.2218498904339627 ...
... 45 ...
... 1.2236746002953676 ...
... 46 ...
... 1.2254475818662858 ...
... 47 ...
... 1.227168915299137 ...
... 48 ...
... 1.2288388101164918 ...
... 49 ...
... 1.2304575793488712 ...
... 50 ...
... 1.2320256189536354 ...
... 51 ...
... 1.2335433913817477 ...
... 52 ...
... 1.2350114124464788 ...
... 53 ...
... 1.2364302407840944 ...
... 54 ...
... 1.2378004693819888 ...
... 55 ...
... 1.2391227187357237 ...
... 56 ...
... 1.240397631303132 ...
... 57 ...
... 1.2416258669854767 ...
... 58 ...
... 1.242808099431435 ...
... 59 ...

... 1.243945012984836 ...
... 60 ...
... 1.2450373001666444 ...
... 61 ...
... 1.2460856595683587 ...
... 62 ...
... 1.2470907940801799 ...
... 63 ...
... 1.2480534093996158 ...
... 64 ...
... 1.2489742127506658 ...
... 65 ...
... 1.2498539117947267 ...
... 66 ...
... 1.2506932136767397 ...
... 67 ...
... 1.2514928242053847 ...
... 68 ...
... 1.2522534471269737 ...
... 69 ...
... 1.2529757834837976 ...
... 70 ...
... 1.2536605310573143 ...
... 71 ...
... 1.2543083838534324 ...
... 72 ...
... 1.2549200316577118 ...
... 73 ...
... 1.2554961596275673 ...
... 74 ...
... 1.256037447926024 ...
... 75 ...
... 1.2565445713936625 ...
... 76 ...
... 1.2570181992492568 ...
... 77 ...
... 1.2574589948187265 ...
... 78 ...
... 1.257867615296539 ...
... 79 ...
... 1.2582447115184938 ...
... 80 ...
... 1.2585909277651655 ...
... 81 ...
... 1.2589069015836747 ...
... 82 ...
... 1.2591932636231997 ...
... 83 ...

```
... 1.2594506374930885 ...
... 84 ...
... 1.2596796396296732 ...
... 85 ...
... 1.25988087918452 ...
... 86 ...
... 1.260054957922832 ...
... 87 ...
... 1.2602024701321866 ...
... 88 ...
... 1.2603240025499618 ...
... 89 ...
... 1.2604201342934758 ...
... 90 ...
... 1.2604914368080846 ...
... 91 ...
... 1.2605384738184295 ...
... 92 ...
... 1.260561801294975 ...
... 93 ...
... 1.2605619674256179 ...
... 94 ...
... 1.2605395125938357 ...
... 95 ...
... 1.2604949693706353 ...
... 96 ...
... 1.260428862507255 ...
... 97 ...
... 1.2603417089382714 ...
... 98 ...
... 1.2602340177878533 ...
... 99 ...
... 1.2601062903878235 ...
```

[50]:
```
i=3
print("Running {}".format(algoNames[i]))
# initialize weights
w_1 = initialize_w(X_train.shape[1], args.w_size)
w_2 = initialize_w(args.w_size,args.w_size)
w_3 = initialize_w(args.w_size, 1)

for j in range(iterations):
    print("......",j,".......")
    start = time.time()
    optimizers = defineOptimizer(i, X_train, y_train, w_1, w_2, w_3, args.lmbda,␣
 ↪args.lr)
    end = time.time()
```

```
    loss_[j,i] = cost(X_train, y_train, w_1, w_2, w_3, args.lmbda)
    ti[j,i] = end-start
    print("......",loss_[j,i],".......")
```

```
Running PGD
... 0 ...
... 1.9321592297883332 ...
... 1 ...
... 1.477266202455614 ...
... 2 ...
... 1.2246516758004224 ...
... 3 ...
... 1.0990579967477265 ...
... 4 ...
... 1.0343790879515156 ...
... 5 ...
... 1.0058903327383446 ...
... 6 ...
... 1.000116262005165 ...
... 7 ...
... 1.006634094022816 ...
... 8 ...
... 1.0167726350502384 ...
... 9 ...
... 1.0306927350688806 ...
... 10 ...
... 1.0453618481688134 ...
... 11 ...
... 1.056431176038997 ...
... 12 ...
... 1.0700855152715645 ...
... 13 ...
... 1.0797012512201631 ...
... 14 ...
... 1.088066473293449 ...
... 15 ...
... 1.094177006019522 ...
... 16 ...
... 1.099336200743627 ...
... 17 ...
... 1.1065780688990547 ...
... 18 ...
... 1.1140990607442898 ...
... 19 ...
... 1.124511604369808 ...
... 20 ...
... 1.1320994512370568 ...
```

... 21 ...
... 1.133433064780415 ...
... 22 ...
... 1.133630757073087 ...
... 23 ...
... 1.1428901352361562 ...
... 24 ...
... 1.1456888785016093 ...
... 25 ...
... 1.149288371145309 ...
... 26 ...
... 1.1536926029487622 ...
... 27 ...
... 1.165907966991599 ...
... 28 ...
... 1.1680380708506073 ...
... 29 ...
... 1.1642083303346284 ...
... 30 ...
... 1.1702103236362291 ...
... 31 ...
... 1.1774158230935643 ...
... 32 ...
... 1.1813172183209792 ...
... 33 ...
... 1.1957662905776274 ...
... 34 ...
... 1.190726059766505 ...
... 35 ...
... 1.1983270112673492 ...
... 36 ...
... 1.206621524356916 ...
... 37 ...
... 1.197770951844999 ...
... 38 ...
... 1.2018677699509595 ...
... 39 ...
... 1.2054911403908959 ...
... 40 ...
... 1.2049691366118862 ...
... 41 ...
... 1.2104213934724328 ...
... 42 ...
... 1.2153361676711492 ...
... 43 ...
... 1.2149488964911768 ...
... 44 ...
... 1.2255535346619308 ...

... 45 ...
... 1.2236389530848861 ...
... 46 ...
... 1.2306508121381858 ...
... 47 ...
... 1.2301541639249922 ...
... 48 ...
... 1.2347038304197984 ...
... 49 ...
... 1.2361902539984932 ...
... 50 ...
... 1.2445402594110182 ...
... 51 ...
... 1.2530190738356697 ...
... 52 ...
... 1.2503351644902718 ...
... 53 ...
... 1.2621033618700501 ...
... 54 ...
... 1.2575578828799587 ...
... 55 ...
... 1.249975813776318 ...
... 56 ...
... 1.2597683167617253 ...
... 57 ...
... 1.2672743230540915 ...
... 58 ...
... 1.2662776645698928 ...
... 59 ...
... 1.2654661435821861 ...
... 60 ...
... 1.2711631358062272 ...
... 61 ...
... 1.2682303568967108 ...
... 62 ...
... 1.259578369064234 ...
... 63 ...
... 1.2657296317848952 ...
... 64 ...
... 1.2640907966115527 ...
... 65 ...
... 1.2842955238603784 ...
... 66 ...
... 1.2908292525393759 ...
... 67 ...
... 1.2896760476921596 ...
... 68 ...
... 1.286106158447178 ...

... 69 ...
... 1.2991018796792664 ...
... 70 ...
... 1.2998787052941985 ...
... 71 ...
... 1.299098397428464 ...
... 72 ...
... 1.3040958906192925 ...
... 73 ...
... 1.3071859791159806 ...
... 74 ...
... 1.3131771692044043 ...
... 75 ...
... 1.318501258268913 ...
... 76 ...
... 1.3252607617315617 ...
... 77 ...
... 1.3247315265882003 ...
... 78 ...
... 1.3242733393983717 ...
... 79 ...
... 1.3197731740555174 ...
... 80 ...
... 1.3178355934537933 ...
... 81 ...
... 1.3157878605131428 ...
... 82 ...
... 1.3207447530085483 ...
... 83 ...
... 1.3257494112059205 ...
... 84 ...
... 1.3157731627784626 ...
... 85 ...
... 1.321189579490124 ...
... 86 ...
... 1.3227781104691216 ...
... 87 ...
... 1.318961654840195 ...
... 88 ...
... 1.3189806469261784 ...
... 89 ...
... 1.315784159844653 ...
... 90 ...
... 1.3099808792119447 ...
... 91 ...
... 1.3022506076913818 ...
... 92 ...
... 1.2959605397035947 ...

```
... 93 ...
... 1.3029860554330914 ...
... 94 ...
... 1.3080332131982308 ...
... 95 ...
... 1.3044323435339589 ...
... 96 ...
... 1.3102421571390195 ...
... 97 ...
... 1.3218087870018356 ...
... 98 ...
... 1.3139022829858737 ...
... 99 ...
... 1.3114125763473157 ...
```

[51]:

```python
i=4
print("Running {}".format(algoNames[i]))
# initialize weights
w_1 = initialize_w(X_train.shape[1], args.w_size)
w_2 = initialize_w(args.w_size,args.w_size)
w_3 = initialize_w(args.w_size, 1)

for j in range(iterations):
    print("......",j,".......")
    start = time.time()
    optimizers = defineOptimizer(i, X_train, y_train, w_1, w_2, w_3, args.lmbda,␣
 ↪args.lr)
    end = time.time()
    loss_[j,i] = cost(X_train, y_train, w_1, w_2, w_3, args.lmbda)
    ti[j,i] = end-start
    print("......",loss_[j,i],".......")
```

```
Running BCD
... 0 ...
... 2.6289867278298633 ...
... 1 ...
... 2.3752888542489257 ...
... 2 ...
... 2.1456202819527843 ...
... 3 ...
... 1.9792222165759907 ...
... 4 ...
... 1.9593088893115783 ...
... 5 ...
... 1.94037431863451 ...
... 6 ...
... 1.844917929115874 ...
```

```
... 7 ...
... 1.7707249716397353 ...
... 8 ...
... 1.7707249716397353 ...
... 9 ...
... 1.7707249716397353 ...
... 10 ...
... 1.711905100967216 ...
... 11 ...
... 1.704202273826422 ...
... 12 ...
... 1.6591863960882394 ...
... 13 ...
... 1.6549376199884767 ...
... 14 ...
... 1.6549376199884767 ...
... 15 ...
... 1.6549376199884767 ...
... 16 ...
... 1.656079571831295 ...
... 17 ...
... 1.615581145240216 ...
... 18 ...
... 1.6121246610037263 ...
... 19 ...
... 1.608842297303887 ...
... 20 ...
... 1.608842297303887 ...
... 21 ...
... 1.608842297303887 ...
... 22 ...
... 1.608842297303887 ...
... 23 ...
... 1.6098708711043848 ...
... 24 ...
... 1.5851937845759592 ...
... 25 ...
... 1.584078056991342 ...
... 26 ...
... 1.56402229643879 ...
... 27 ...
... 1.56402229643879 ...
... 28 ...
... 1.547599771739899 ...
... 29 ...
... 1.546999528256799 ...
... 30 ...
... 1.5338483264601717 ...
```

```
... 31 ...
... 1.5338483264601717 ...
... 32 ...
... 1.5338483264601717 ...
... 33 ...
... 1.523033157763402 ...
... 34 ...
... 1.5144246991060528 ...
... 35 ...
... 1.5144246991060528 ...
... 36 ...
... 1.5072286216116428 ...
... 37 ...
... 1.501290459254117 ...
... 38 ...
... 1.4974464354940555 ...
... 39 ...
... 1.4933969062170753 ...
... 40 ...
... 1.4900522990229756 ...
... 41 ...
... 1.4887271839536456 ...
... 42 ...
... 1.4902278050906679 ...
... 43 ...
... 1.4902278050906679 ...
... 44 ...
... 1.4909175449690095 ...
... 45 ...
... 1.4909175449690095 ...
... 46 ...
... 1.49241156784851 ...
... 47 ...
... 1.4914492276914864 ...
... 48 ...
... 1.4921105916570518 ...
... 49 ...
... 1.4936399684500175 ...
... 50 ...
... 1.4936399684500175 ...
... 51 ...
... 1.4942907429045937 ...
... 52 ...
... 1.4949368744616207 ...
... 53 ...
... 1.4949368744616207 ...
... 54 ...
... 1.4946266857797323 ...
```

```
... 55 ...
... 1.4961849951421056 ...
... 56 ...
... 1.494436987951873 ...
... 57 ...
... 1.4960089155367209 ...
... 58 ...
... 1.4960089155367209 ...
... 59 ...
... 1.4944428735702442 ...
... 60 ...
... 1.4944428735702442 ...
... 61 ...
... 1.4944428735702442 ...
... 62 ...
... 1.4937428311321184 ...
... 63 ...
... 1.4943235397571961 ...
... 64 ...
... 1.4934795106334486 ...
... 65 ...
... 1.4940471592130393 ...
... 66 ...
... 1.4940471592130393 ...
... 67 ...
... 1.4940471592130393 ...
... 68 ...
... 1.4934621252541695 ...
... 69 ...
... 1.4940171297893299 ...
... 70 ...
... 1.4936455288073294 ...
... 71 ...
... 1.4950410125267064 ...
... 72 ...
... 1.4951628597054976 ...
... 73 ...
... 1.4968394338445963 ...
... 74 ...
... 1.4973523557797728 ...
... 75 ...
... 1.4973523557797728 ...
... 76 ...
... 1.4990070855199267 ...
... 77 ...
... 1.499510973737929 ...
... 78 ...
... 1.5000107436273995 ...
```

```
... 79 ...
... 1.5002777254493396 ...
... 80 ...
... 1.5002777254493396 ...
... 81 ...
... 1.5019400436335137 ...
... 82 ...
... 1.5016921656485493 ...
... 83 ...
... 1.5016921656485493 ...
... 84 ...
... 1.5033070757897598 ...
... 85 ...
... 1.5033070757897598 ...
... 86 ...
... 1.5033070757897598 ...
... 87 ...
... 1.5033070757897598 ...
... 88 ...
... 1.5048819973786072 ...
... 89 ...
... 1.5043633903257785 ...
... 90 ...
... 1.5043633903257785 ...
... 91 ...
... 1.5059024117088786 ...
... 92 ...
... 1.5059024117088786 ...
... 93 ...
... 1.5057690114012705 ...
... 94 ...
... 1.5072924320472925 ...
... 95 ...
... 1.5072924320472925 ...
... 96 ...
... 1.5072924320472925 ...
... 97 ...
... 1.5072924320472925 ...
... 98 ...
... 1.5072924320472925 ...
... 99 ...
... 1.5077272793978247 ...
```

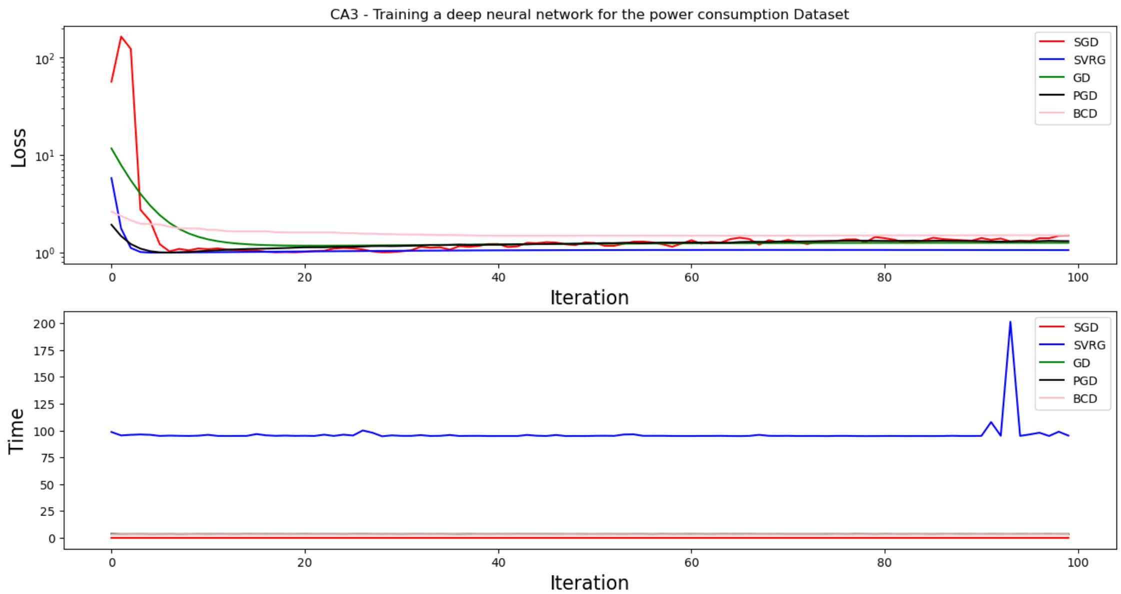
[83]:

```
# Plot results
# Define plotting variables
fig, ax = plt.subplots(2, 1, figsize=(16, 8))
ax[0].legend(loc="upper right")
```

```
ax[0].set_xlabel(r"Iteration", fontsize=16)
ax[0].set_ylabel("Loss", fontsize=16)
ax[0].set_title("CA3 - Training a deep neural network for the power consumption␣
 ↪Dataset")
#ax[0].set_ylim(ymin=0)
ax[0].set_yscale('log')
ax[0].plot(loss_[:,0], color="red")
ax[0].plot(loss_[:,1], color="blue")
ax[0].plot(loss_[:,2], color="green")
ax[0].plot(loss_[:,3], color="black")
ax[0].plot(loss_[:,4], color="pink")
ax[0].legend(['SGD', 'SVRG', 'GD', 'PGD', 'BCD'])
#plt.show()
#
ax[1].legend(loc="upper right")
ax[1].set_xlabel(r"Iteration", fontsize=16)
ax[1].set_ylabel("Time", fontsize=16)
#ax[1].set_ylim(ymin=50, ymax=100)
#ax[1].set_yscale('log')
ax[1].plot()
ax[1].plot(ti[:,0], color="red")
ax[1].plot(ti[:,1], color="blue")
ax[1].plot(ti[:,2], color="green")
ax[1].plot(ti[:,3], color="black")
ax[1].plot(ti[:,4], color="pink")
ax[1].legend(['SGD', 'SVRG', 'GD', 'PGD', 'BCD'])
plt.show()
plt.savefig("CA3-power.png")

#plt.savefig("power.png")
#plt.savefig("GHG.png")
```

```
No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
No artists with labels found to put in legend.  Note that artists whose label
start with an underscore are ignored when legend() is called with no argument.
```



<Figure size 640x480 with 0 Axes>

[ ]: